27 | 多任务环境中的Java性能问题，怎样才能不让程序互相干扰？

2020-01-27 庄振运

性能工程高手课 进入课程 >

讲述：庄振运
时长 20:44　大小 16.62M

你好，我是庄振运。

我们来继续学习生产实践中的案例。在生产实践中，为了降低公司运营成本，更好地利用系统容量，并提高资源使用率，我们经常会让多个应用程序，同时运行在同一台服务器上。

但是，万事有利就有弊。这几个共存的应用程序，有可能会互相影响；有时还会导致严重的性能问题。我就遇到过，几个程序同时运行，最后导致吞吐量急剧下降的情况。

所以，今天我们就来探讨，当多个 Java 应用程序共存在一个 Linux 系统上的时候，会产生哪些性能问题？我们又该怎么解决这些问题？

怎样理解多程序互相干扰?

为了更好地理解后面的性能问题，你需要先了解一下应用程序内存管理机制的背景知识。我们运行的是 Java 程序，所以先快速复习一下 **Java 的 JVM 内存管理机制**。

Java 程序在 Java 虚拟机 JVM 中运行，JVM 使用的内存区域称为**堆**。JVM 堆用于支持动态 Java 对象的分配，并且分为几个区域，称为“代”（例如新生代和老年代）。Java 对象首先在新生代中分配；当这些对象不再被需要时，它们会被称为 GC（Garbage Collection）的垃圾回收机制收集。发生 GC 时，JVM 会从根对象开始，一个个地检查所有对象的引用计数。如果对象的引用计数降为零，那就删除这个对象，并回收使用这个对象相应的存储空间。

GC 运行的某些阶段，会导致应用程序停止响应其他请求，这种行为，通常称为 STW（Stop The Word 暂停）。 JVM 调优的重要目标之一，就是最大程度地减少 GC 暂停的持续时间。

复习完 JVM 内存管理机制，我们还要看一下与它相关的 **Linux 的内存管理机制**。

在 Linux 操作系统上，虚拟内存空间基本上是固定大小（例如 4KB）的页面。Linux 近年来有很多内存管理的优化，来提高内存使用效率和运行进程的性能。

Linux 内存管理有一个**页面回收**的机制。它在内部维护一个空闲页面（Free Page）列表，来满足未来应用程序的内存请求。当空闲页面的数量下降到一定水平时，操作系统就开始回收页面，并将新回收的页面添加到空闲列表中。

执行页面回收时，操作系统需要进行页面扫描（Page Scanning），以检查已经分配页面的活动性。Linux 有两个策略来进行页面扫描：**后台扫描**（由 kswapd 守护程序执行）和**前台扫描**（由进程自己执行）。

通常情况下，后台扫描就够了，应用程序的性能一般不会受到影响。但是当操作系统的内存使用非常大，空闲页面严重不足时，Linux 就会启动前台页面回收，也被称为**直接回收或同步回收**。在前台页面回收过程中，应用程序会停止运行，因此对应用程序影响很大。

Linux 还有一个**页面交换**（Page Swapping）的机制。是当可用内存不足时，Linux 会将某些内存页面换出到外部存储，以回收内存空间来运行新进程。当对应于换出页面的内存空

间，再次处于活动状态时，系统会把这些页面重新从外部存储换入内存。

内存管理方面，THP（Transparent Huge Pages，透明大页面）是另外一个机制，也是为了提高进程的性能。我们在第 22 讲讨论过，如果系统用较大的页面，比如 2MB，而不是传统的 4KB，那么会带来一些好处，尤其是所需的地址转换条目数会减少。

尽管使用大页面的好处很早就为人所理解，但在 THP 引入之前，程序想使用大型页面并不容易。例如，操作系统启动时，需要保留大页面，并且进程必须显式调用才能分配大页面。而 THP 就是为了避免这两个问题而设计的，因此操作系统默认情况下就启用 THP。

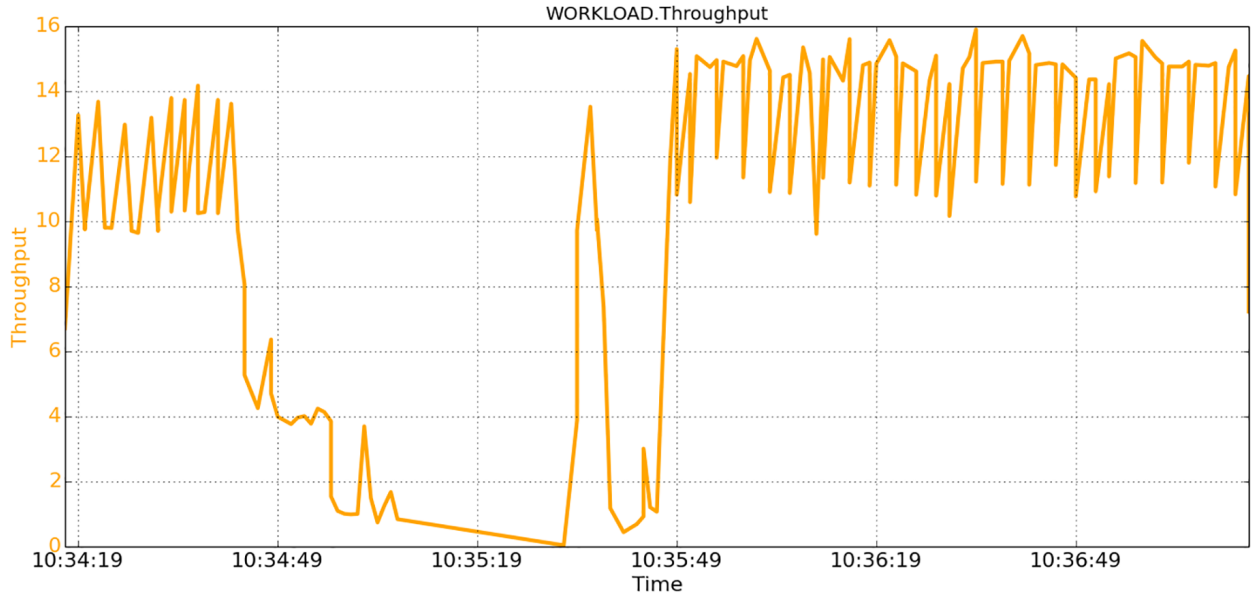
了解了背景知识，你再看多个应用程序共存时的两个场景就不会有障碍了。第一个场景是应用程序启动时，第二个场景是应用程序稳定运行时。

应用程序启动时为什么会被其他程序干扰？

我们先看应用程序启动时的场景。当几个共存的应用程序共享有限的计算资源（包括内存和 cpu）时，它们之间会相互影响。如果各自独立地运行，导致对系统计算资源的消耗无法协调一致，那么某些应用程序会出现问题。

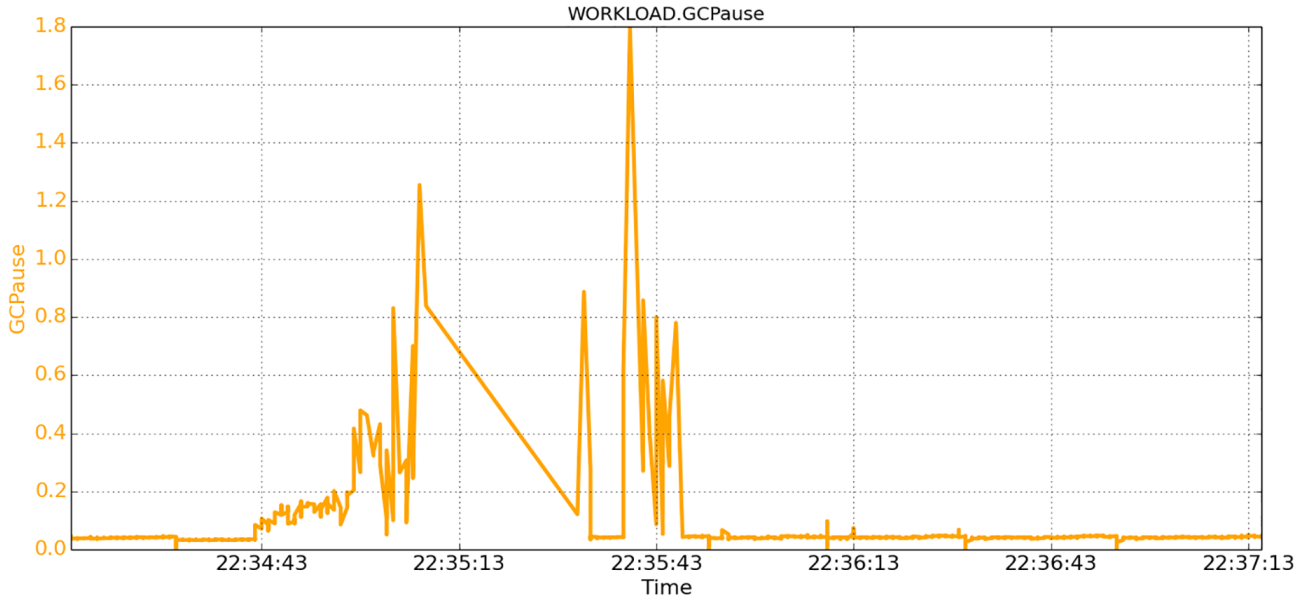
我们要做个实验来暴露这些性能问题，看看这个问题的表象是什么，然后一起分析产生问题的原因。

这个实验采用了两个相同的 Java 程序。我们首先启动第一个程序，来占用一些内存，系统剩下约 20GB 的未使用内存。然后我们开始启动另外一个 Java 程序，这个程序需要 20GB 的堆。



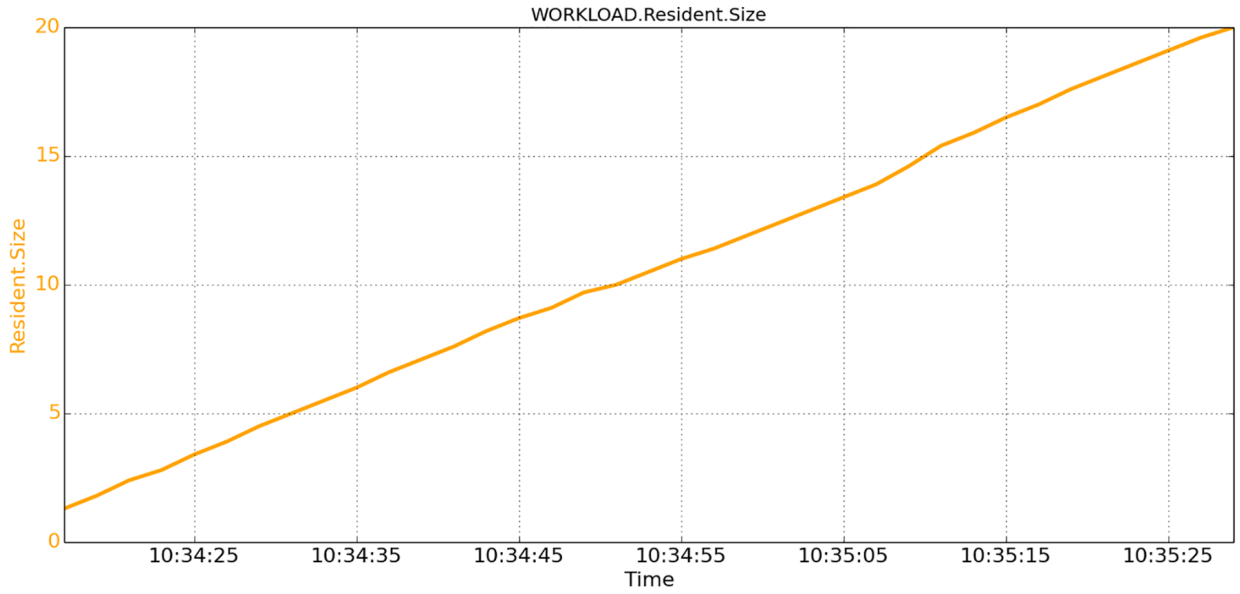
在图中你可以看到，启动第二个程序之后，它的吞吐量是 12K/ 秒，持续时间约 30 秒。然后，吞吐量开始急剧下降。最坏的情况，在大约 20 秒的时间内，吞吐量几乎为零。有趣的是，过了一会儿，吞吐量又再次回到了稳定状态：12KB/ 秒。

下图显示了同一时间段的 GC 暂停信息。



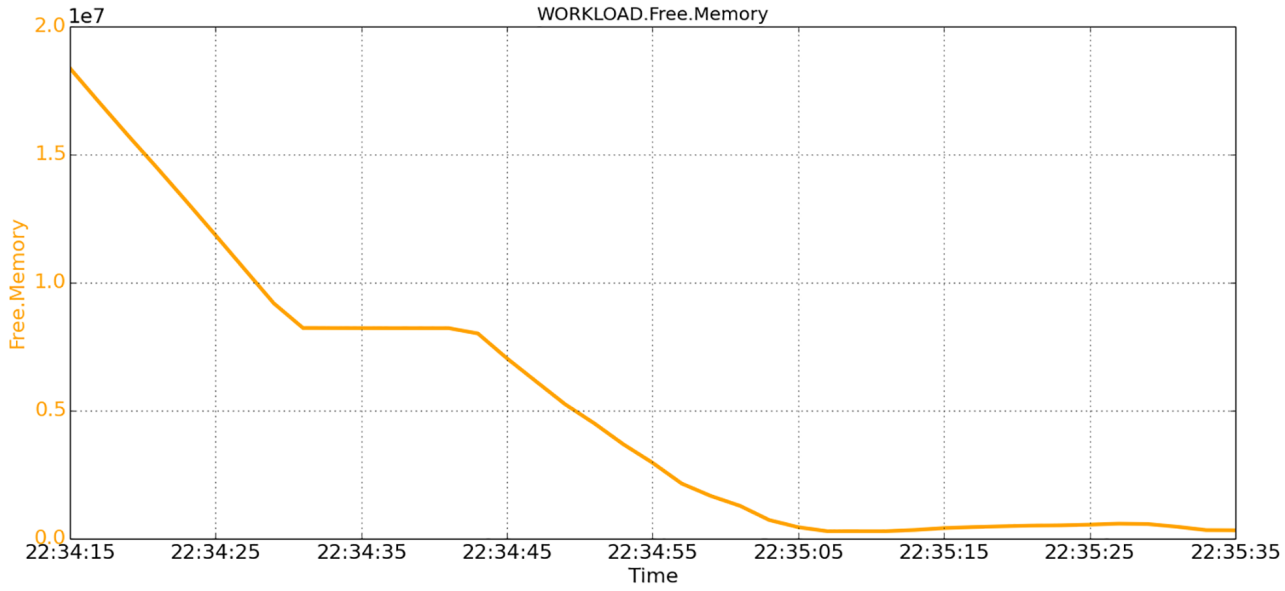
最初的 GC 暂停很低，都低于 50 毫秒；然后暂停就跳到数百毫秒之大。你甚至可以看到两次大于 1 秒的超大的暂停！大约 1 分钟后，GC 暂停再次下降至低于 50 毫秒，并变得稳定。

我们看到在启动期间，Java 程序的性能很差。因为问题是在启动 JVM 时发生的，我们有理由怀疑这与 JVM 的启动方式有关。我们检查了程序的的内存驻留大小（RES，Resident Size），也就是进程使用的未交换的物理内存，图示如下。

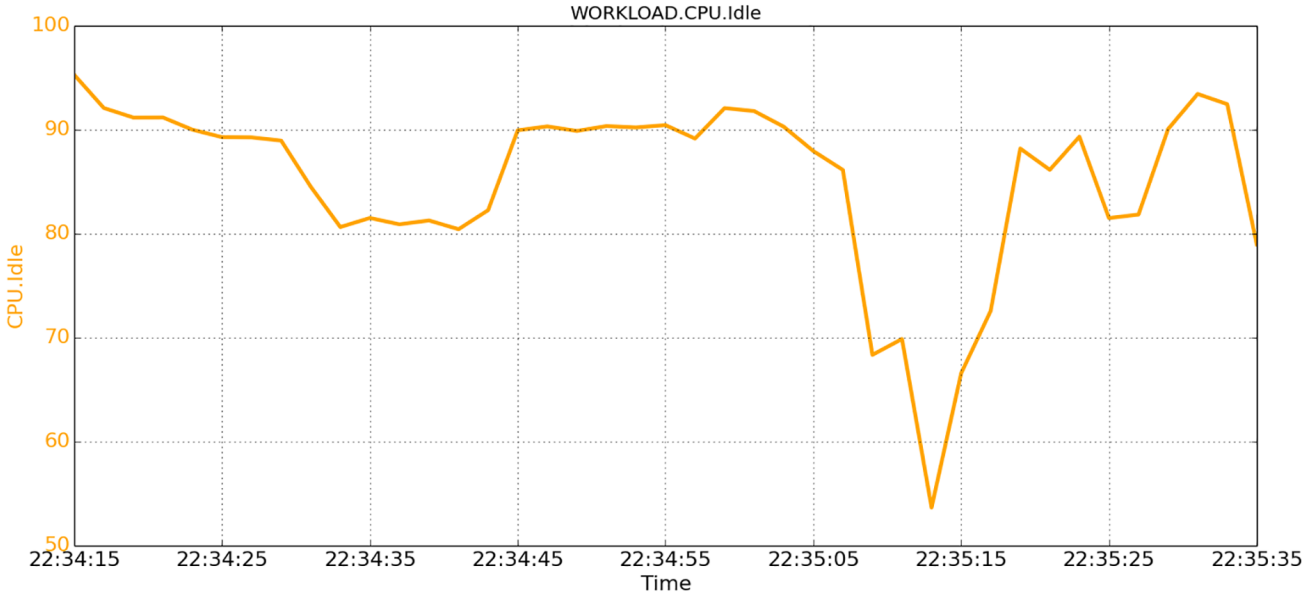


从图中你可以看到，尽管我们在启动 JVM 时，用参数将 JVM 的堆大小指定为 20GB（-Xmx20g 和 -Xms20g），但是 JVM 并不会从内存中一次全部拿到 20GB 的堆空间。相反，操作系统会在 JVM 的运行过程中不断地分配。也就是说，随着 JVM 实例化越来越多的对象，JVM 会从操作系统逐渐拿到更多的内存页面来容纳它们。

在分配过程中，操作系统将不断地检查空闲页面列表。如果发现可用内存量低于一定水平，操作系统就会开始回收页面，这个过程会花费 CPU 的时间。根据可用内存短缺的严重程度，回收过程可能会严重阻塞应用程序。在下图中，我们看到，可用内存明显地下降到了非常低的水平。



下面这张图显示了 CPU 的空闲百分比（CPU 空闲百分比和繁忙百分比的和是 100%）。对比时间线，我们可以清楚地看到，页面回收过程会导致 CPU 开销，也就是空闲百分比下降了。

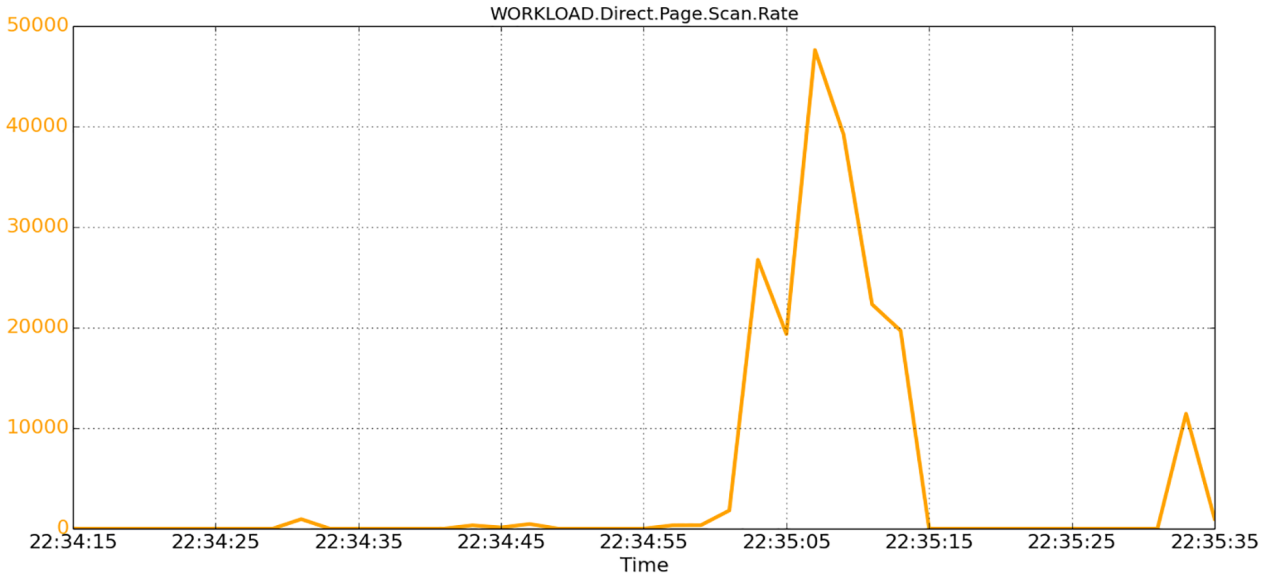


那么怎么进行内存回收呢?

在 Linux 上，当可用内存不足时，操作系统会唤醒 **kswapd 守护程序**，开始在后台回收空闲页面。如果内存压力很大，操作系统就会被迫采取另外一种措施，就是**直接地同步释放内存的前台**。具体来讲，当可用空闲页面降到一个阈值之下，就会触发这种直接前台回收。

当发生直接前台回收时，Linux 会冻结正在申请内存的执行代码的应用程序，从而间接地导致大量的 GC 暂停。

此外，直接回收通常会扫描大量内存页面，以释放未使用的页面。那么我们就来看看 Linux 直接回收内存页面的繁忙程度。下图就画出了 Linux 通过直接回收路径扫描的页面数。



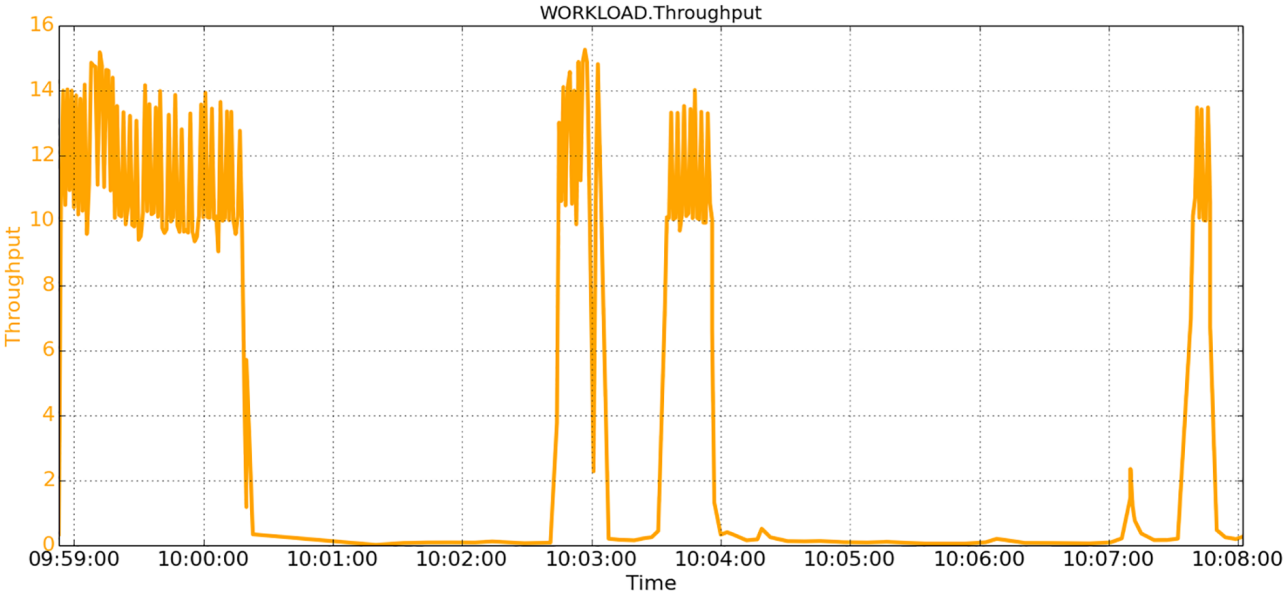
我们看到，在峰值时，通过直接回收，每秒扫描约 48K 个页面（即 200 MB）；这个回收工作量是很大的，CPU 会不堪重负。

运行中的应用程序为什么会被别的程序干扰?

了解过程序启动时互相干扰的场景，我们再来考虑第二个场景：应用程序在持续运行中。

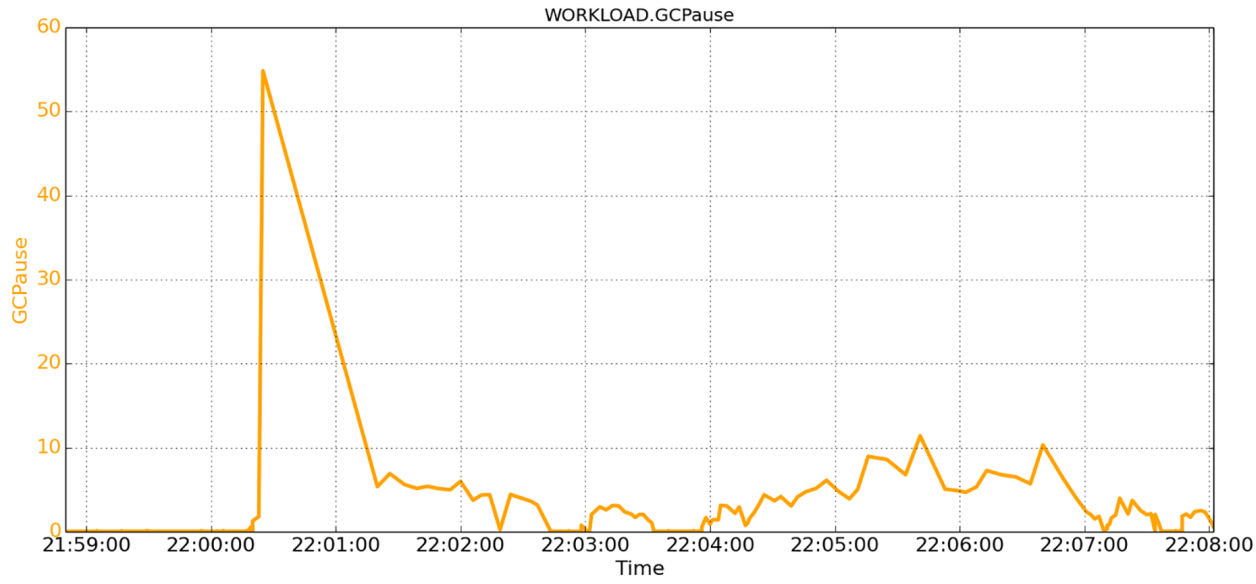
我们的实验是这样进行的。第一个 Java 程序以 20GB 的堆启动，并进入稳定状态。然后另外一个程序启动，并开始分配 50GB 的内存。

下图中体现了第一个程序的吞吐量。



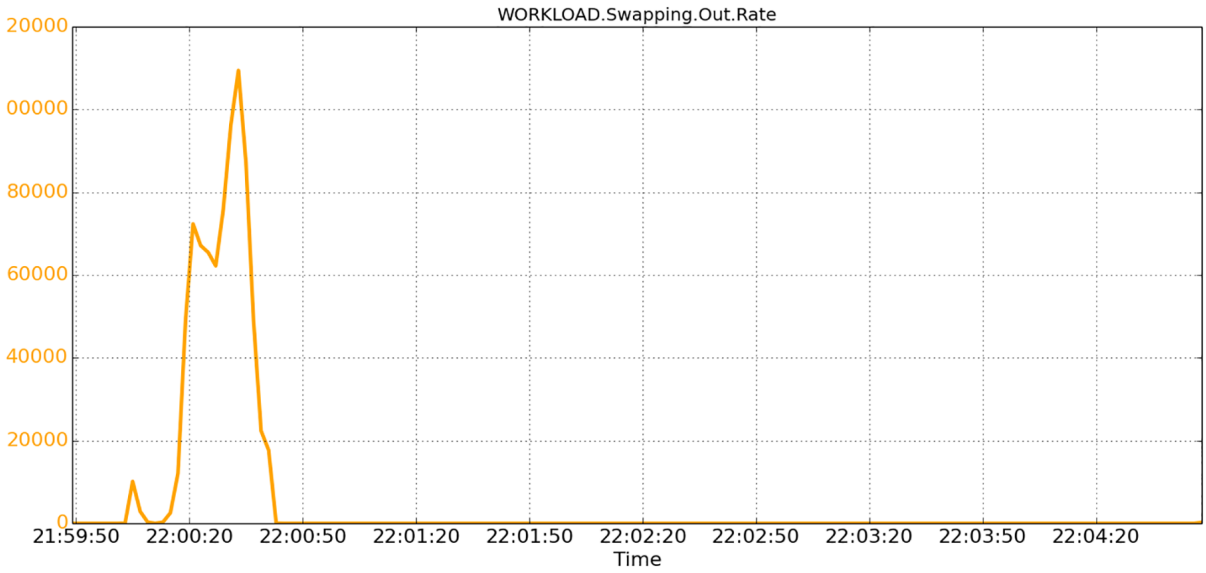
从图中我们看到，第一个程序从一开始就实现了稳定的 12K/ 秒的吞吐量。然后，吞吐量急剧下降到零，这个零吞吐量的过程持续了约 2 分钟。从那时起，吞吐量一直在发生相当大的变化：有时吞吐量是 12K/ 秒，其他时候又降为零。

我们也观察了 JVM 的暂停，用下图所示。



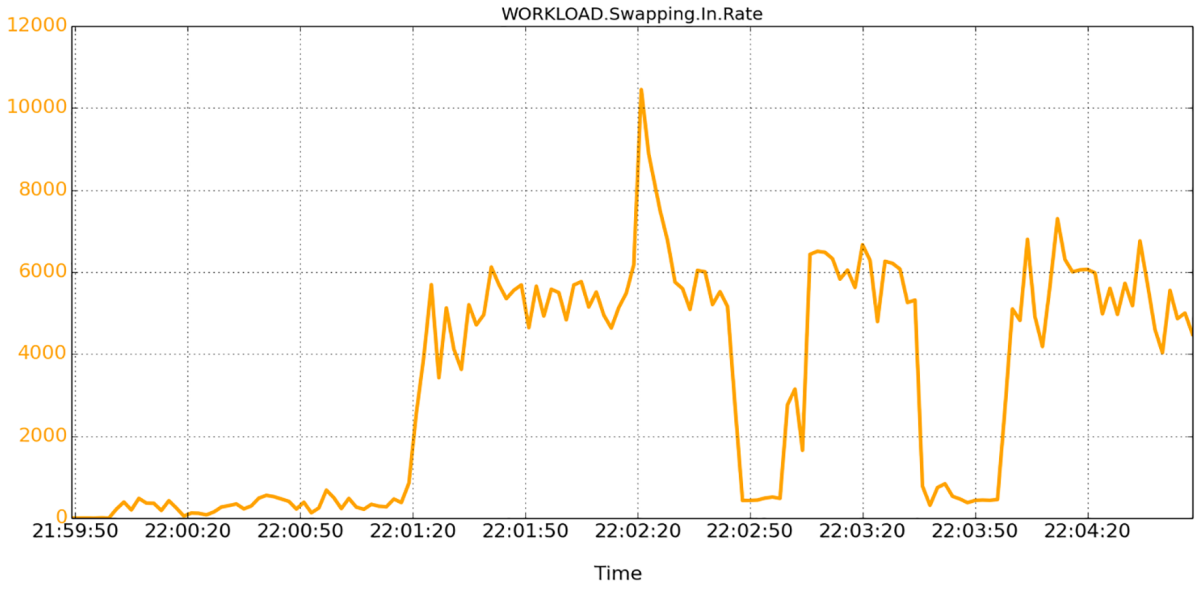
从图中我们看到，在稳定状态下，GC 暂停几乎为零，然后居然有一个超级大的暂停；多大呢？55 秒！从那时起，GC 暂停持续变化，但很少恢复为零。大多数暂停时间为几秒钟。

我们观察到，其他应用程序的运行会严重影响本程序的性能。各种观察的结论是，系统处于内存压力之下，操作系统内存会有很多和外部存储的页面交换活动。在下图中，我们看到操作系统交换出了很多内存页面到外部存储空间。



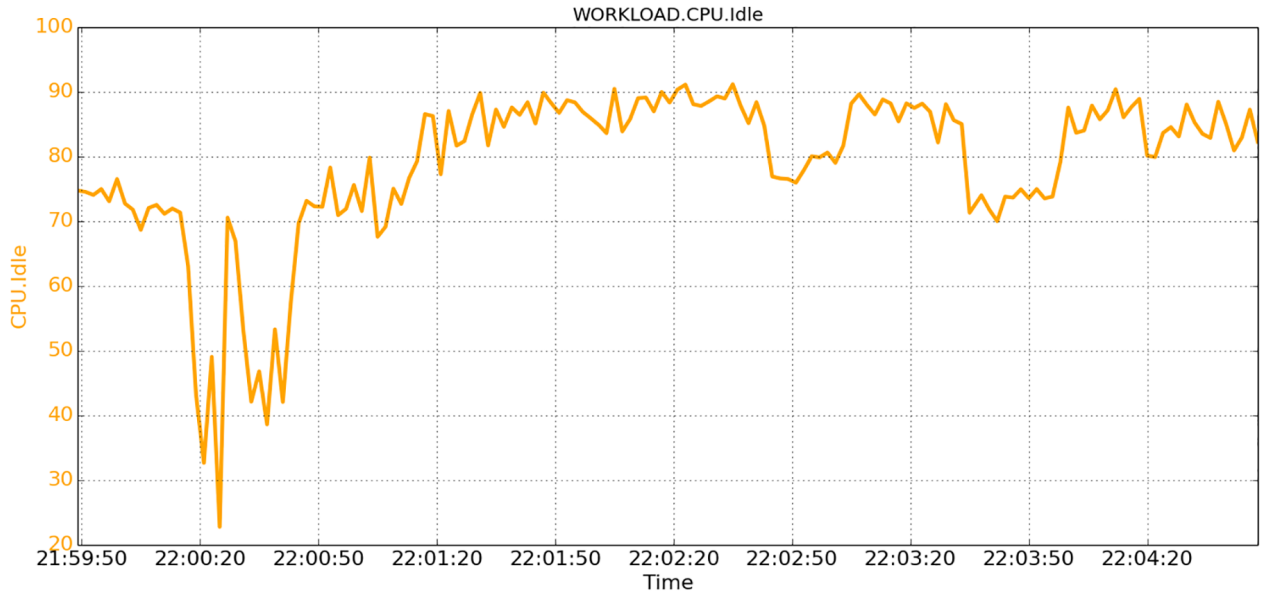
这些换出的内存页面很多属于 Java 程序（也就是堆空间）。如果 JVM 需要进行堆上的垃圾回收，也就是 GC，那么 GC 需要扫描 JVM 对象，以收集失效的对象。如果扫描的对象恰好是分配在换出的页面上，那么 JVM 需要先将它们从外部存储交换空间重新载入到内存中。从外部存储载入内存需要一些时间，因为交换空间通常位于磁盘驱动器上。

所有这些时间，都会算在 GC 暂停之中。因此，程序会看到较大的 GC 暂停。下图就显示了大量的从外部存储载入页面的活动。



尽管页面交换活动会增加 GC 暂停时间，似乎可以解释刚刚看到的 JVM 暂停。但是，我怀疑，仅是这个原因根本无法解释生产中看到的很大暂停，比如超过 55 秒的暂停。你可能会问，我为什么有这样的怀疑？因为我在许多 GC 暂停的过程中，观察到了较高的系统 CPU 使用。

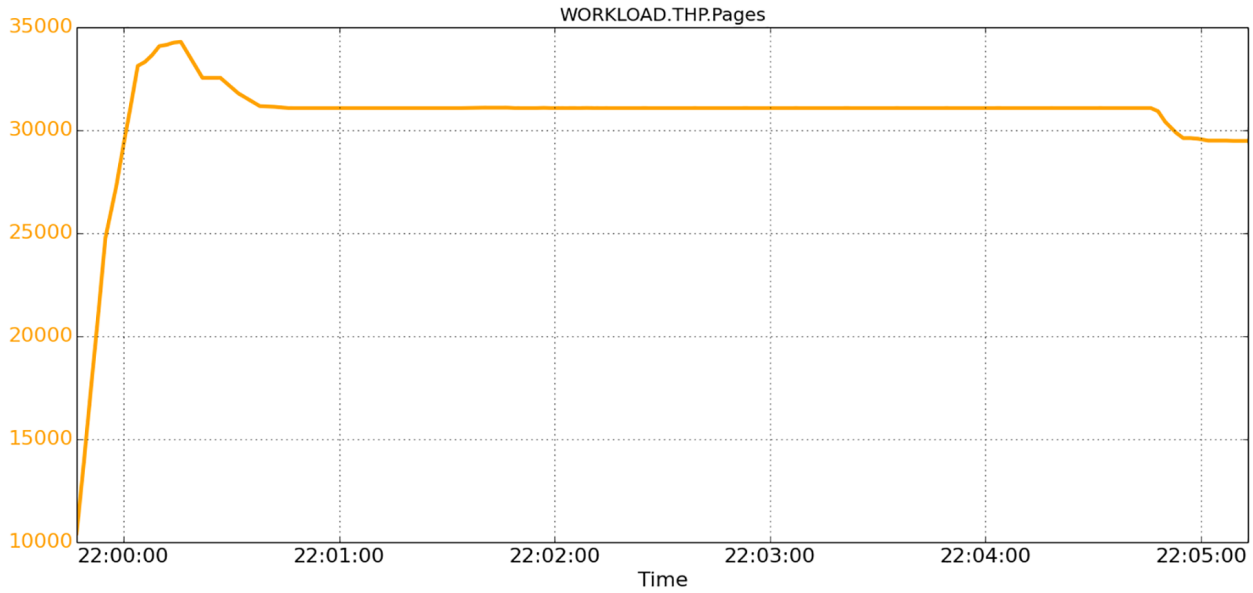
比如在下图中，我们观察到，系统也处于严重的 CPU 压力下。



CPU 的高使用率不能完全归因于页面交换活动，因为页面交换通常不会占用大量 CPU。所以，其中“必有隐情”：一定是有其他活动在大量使用 CPU。我们通过检查了各种系统性

能指标，最终确定了根因：是由于 THP 的机制，该机制严重加剧了程序性能和系统性能的下降。

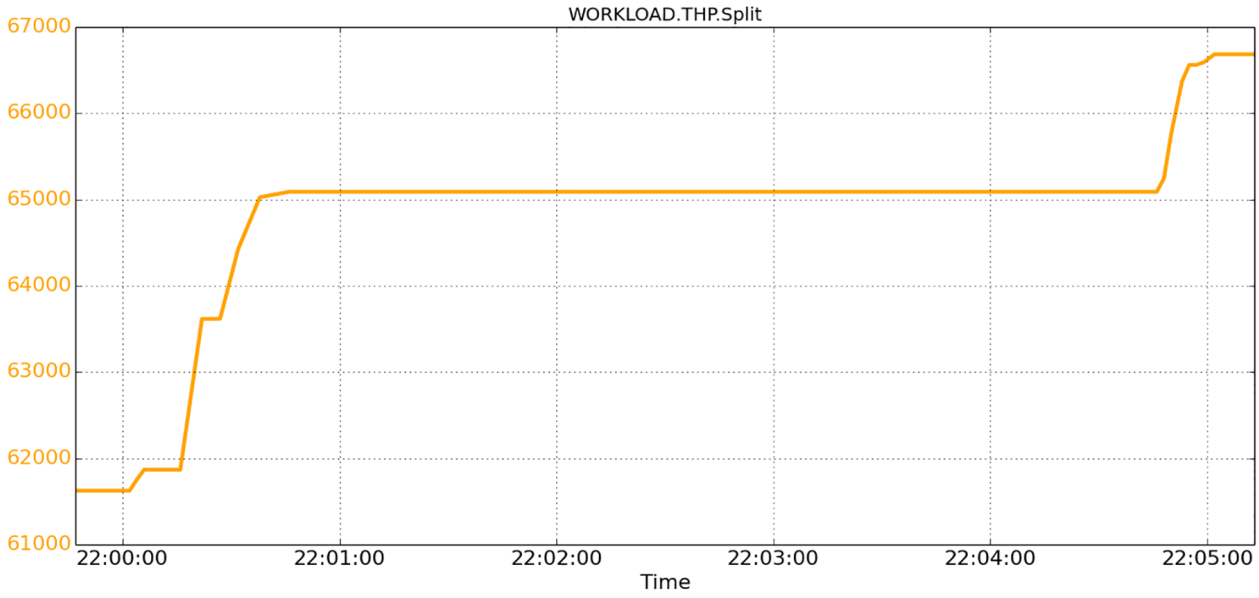
具体来说，Linux 启用 THP 后，当应用程序分配内存时，会优先选择 2MB 大小的透明大页面，而不是 4KB 的常规页面。这一点我们可以轻易验证，比如下图中显示了透明大页面的瞬时数量。在峰值时，我们看到约 34,000 个 THP，即约 68GB 的内存量。



我们还观察到，THP 的数量一开始很高，一段时间后开始下降。这是因为某些 THP 被拆分成小的常规页面，以补充可用内存的不足。

为什么需要拆分大页面呢？是因为当 Linux 在有内存压力时，它会将 THP 分为常规的、要准备交换的页面。为什么必需拆分大页面？这是因为当前的 Linux，仅支持常规大小页面的交换。

拆分活动的数量我们也用下图画出来了。你可以看到，在五分钟内大约有 5K 个 THP 页面被拆分，对应于 10GB 的内存。



除了大页面拆分，同时，Linux 也会尝试将常规页面重新聚合为 THP 大页面，这就需要额外的页面扫描，并消耗 CPU。如果你在实践中注意观察的话，可以发现这种活动会占用大量 CPU。

使用 THP 可能遇到的更糟糕的情况是，**聚合**和**拆分**这两个相互矛盾的活动，是来回执行的。也就是说，当系统承受内存压力时，THP 被拆分成常规页面，而不久之后，常规页面则又被聚合成 THP，依此类推。我们已经观察到，这种行为会严重损害我们生产系统中的应用程序性能。

如何解决多程序互相干扰?

那么程序在启动和运行时互相干扰的性能问题，到底该怎么解决呢？我们现在就来看解决方案。

我们的解决方案由三个设计元素组成，每个设计元素都针对问题的特定方面。部署任何单独的元素都将在一定程度上对问题有所帮助。但是，所有设计元素协同工作，才能获得最好的效果。

第一个设计元素是**预分配 JVM 的堆空间**。

我们知道，对 JVM 而言，只有在实际使用堆空间之时，就是当需要增大堆空间来容纳新对象分配请求时，Linux 才会为之分配新的内存页面，这时就可能会触发大量页面回收，并损

害程序和系统性能。

这个设计元素就是预分配所有堆空间，从而避免了 Linux 实时分配页面的不利场景。要预先执行堆预分配，需要使用一个特殊的 JVM 参数：“-XX：+ AlwaysPreTouch”，来启动 Java 应用程序。

但这个设计元素也有副作用，就是增加了 JVM 启动所需的时间，在部署时你需要考虑这一点。我们也做过一些实际测量，这个额外启动时间并不大，一般在几秒钟内，通常是可以接受的。

第二个设计元素，是关于如何**保护 JVM 的堆空间不被唤出到外部存储**。

我们知道，当发生 GC 时，JVM 需要扫描相应的内存页。如果这些页面被操作系统换出到外部存储，则需要先换入它们到内存，这就会导致延迟，会增加 JVM 的暂停时间。

这个设计元素就可以防止 JVM 的堆页面被换出。 我们知道，Linux 操作系统上是可以关闭内存页面交换的，但是这个设置如果是在系统级别进行，就会影响所有应用程序和所有内存空间。我推荐你一个更好的实现，就是采用**微调**，你来选择哪个应用程序和哪个存储区域可以页面交换。例如，你可以使用 cgroup 来精确控制要交换的应用程序。

公司中的大多数平台，一般都用来运行同类 Java 应用程序；这些程序往往配置差不多。在这些情况下，在系统级别关闭应用程序交换，倒也是非常合理的。

第三个设计元素是**动态调整 THP**。

我们已经看到，启用 THP 功能可能会在某些场景下，导致严重的性能损失；但是 THP 在其他场景的确提高了性能，所以到底是否要启用 THP 呢？我们需要仔细考虑。

当 THP 影响性能时，系统的可用内存往往也恰好严重不足。发生这种情况时，现有的 THP 需要拆分成常规页面以进行页面换出。所以，我建议你用一个可用内存大小的阈值来决定 THP 的开关。

具体来说，就是建议你使用**应用程序的堆大小**作为内存阈值，来决定是否打开或关闭 THP。当可用内存远远大于应用程序的内存可能占用量大小时，就启用 THP，因为系统不

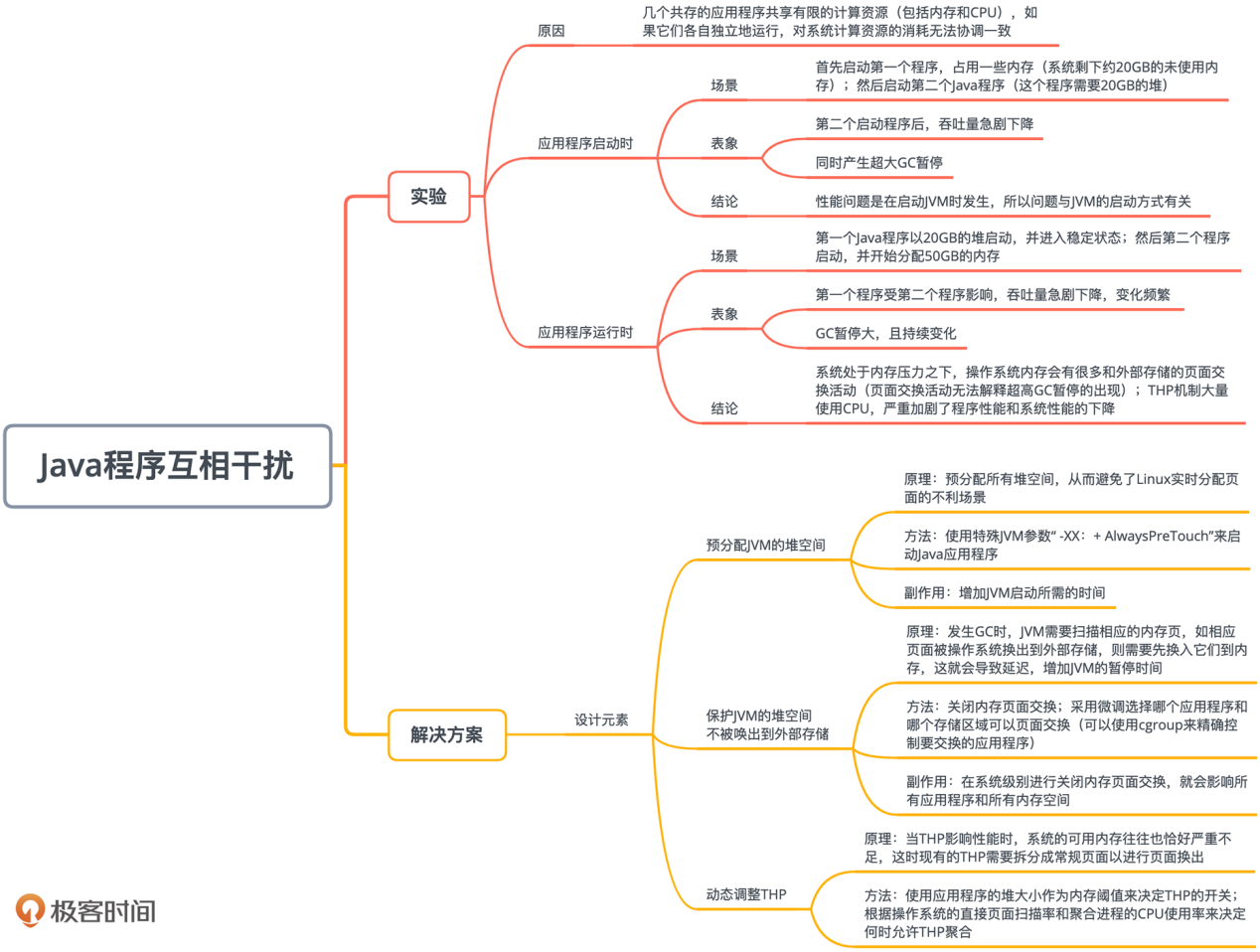
太可能在启动特定应用程序后出现内存压力。否则的话，就关闭 THP。

由于许多后端服务器都是运行同类应用程序，通常情况下，你都很容易知道，部署的应用程序预期会占用多少内存空间。

此外，常规页面需要聚合成 THP，才能将大页面分配给应用程序。因此，这个元素的另外一部分机制是进行微调，是决定何时允许 THP 聚合。我建议你根据**操作系统的直接页面扫描率**和**聚合进程的 CPU 使用率**来决定。

总结

今天我们讲述了，将多个应用程序放置在同一台服务器上时，由于应用程序和操作系统机制的互相作用，引发的一系列性能问题。这些问题的根本原因，就是程序之间的互相影响。



应用程序之间的关系和人际关系一样，有时和谐，有时不和谐。唐代诗人刘禹锡有几句诗说：“常恨言语浅，不如人意深。今朝两相视，脉脉万重心。”说的是，语言的表达能力通

常很有限，所以两人只能用眼神传达更复杂的情感。应用程序之间的关系，甚至程序和操作系统及硬件之间的关系，也会很复杂，也需要做足够的性能分析，才能理清它们之间的关系。

今天的讲述，主要集中在多任务共存环境中的两个问题，重点在分析问题产生的复杂根因。如果你对这方面的具体算法和生产验证有兴趣，可以参考我的一篇论文。这篇论文发表在International Journal of Cloud Computing上面。

思考题

Linux 操作系统的 THP 机制的设计初衷，本是为了提升系统性能。可是在有些情况下反而导致了系统性能下降。想一想操作系统的其他机制，有没有类似的情况发生？

> Tips：文件系统的预先读取等。

欢迎你在留言区分享自己的思考，与我和其他同学一起讨论，也欢迎你把文章分享给自己的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (3)

写留言

黄海峰
2020-01-28

这篇干货了，以前看过好多关于jvm调优的文章或教程都没有提到这些知识点。。

1

kyleqian
2020-01-31

请问文中说的程序吞吐量是什么意思？比如那个12K/秒，具体指的什么？是分配内存的调用吗？另外THP可以随时打开或关闭吗？我以为只能在系统启动时设定呢

展开

Pasta
2020-01-28

老师讲的非常赞，又重温了Linux的page reclaim部分。学到了好多，比如何时控制THP的开关，以及THP的缺点。习题猜测是文件系统预先读取很多小文件时有影响，比如小于4kB的文件，而通常预读有128kB，可能会造成浪费，比如大量读写磁盘，导致D状态，影响性能？不过只是猜测，希望到时候看到老师的统一习题解答:)

展开